Taming the SHARC™*

Maximizing SHARC Performance
by
Understanding and Exploiting SHARC Features

By Tim Cooper
Ixthos, Inc.

The Analog Devices SHARC is a powerful computing device. However, a proper understanding of its architectural features is required to optimize performance. It is easy to program the SHARC. It is challenging to program the SHARC well. To program the SHARC well requires an understanding of its many features and their attendant advantages.

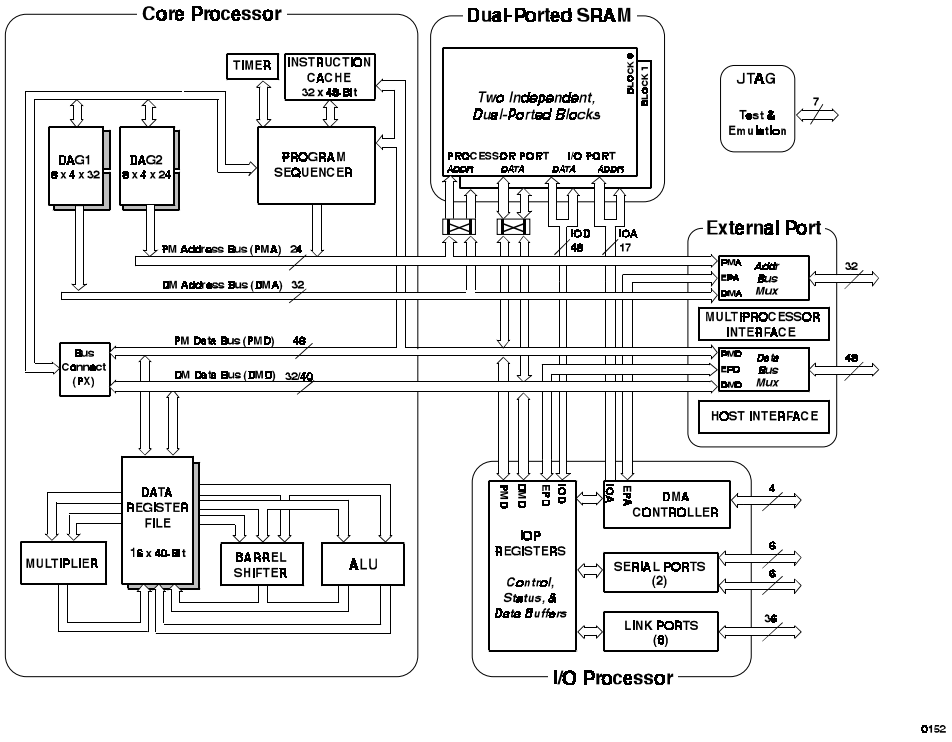

Figure 1. SHARC Internal Architecture.[1]

I. Performance Features Common to DSP and General Purpose Processors.

Digital Signal Processing (DSP) is characterized by repetitive numerical computations. High speed, low overhead data movement and rapid computation are required to attain high performance in the DSP realm. These areas are not emphasized to the same degree in traditional embedded control applications, which are more reliant upon features that minimize interrupt latency. Therefore, features that specifically optimize

* SHARC is a registered trademark of Analog Devices, Incorporated.

computations provide a substantial pay-off in the world of digital signal processing, while these features provide less benefit in generic embedded applications.

The SHARC possesses several architectural features that are not specifically related to numerical computational performance. These features provide good performance in both the DSP and the general purpose processing realms. Many of these features have been around for decades; only in recent years has technology been able to economically integrate these features into a single computing device. Features that provide significant performance advantages in both the signal processing and general purpose computing realms are the following[2]:

Pipelining improves performance by processing instructions in discrete stages, like an assembly line. With each clock cycle, instructions propagate through a pipeline. The SHARC uses a pipeline having three stages: instruction fetch, decode, and execute. It therefore takes three clock cycles for an instruction to propagate through the pipe and complete execution. Because each stage of the pipeline can work on an instruction independently, a full pipe generates instruction result with each clock cycle. The processor execution speed is one instruction per clock cycle, even though each instruction requires three clock cycles.

Delayed Branches are a common way to recover clock cycles in a pipelined machine. A delayed branch is best understood by considering the behavior of a pipelined processor when it executes a branch instruction. When the branch instruction gets to the execution stage of the pipeline, the processor will begin to fetch instructions from a new location in the program. Unfortunately, the processor has already fetched and started decoding the two instructions that follow the branch. When the processor takes the branch, it must discard these two instructions, and re-load the pipeline starting from the branch target address.

This pipeline reload means that two instruction fetches are wasted. This may not seem significant, but consider a loop that copies data. The loop may have one, two, or a few instructions in the loop body depending upon the machine instruction set. The act of branching back to the top of the loop will waste two clock cycles per loop iteration. The ratio of the wasted clock cycles to the few instructions in the loop body is significant. If these wasted clock cycles can be made useful, performance will improve substantially.

The SHARC provides a delayed branch instruction that recovers these clock cycles. A delayed branch is a transfer of control instruction (such as a call, return, or a conditional or unconditional jump) that does not take effect until the two instructions following the branch have been executed. This keeps the pipeline full.

The programmer new to SHARC assembly language may find it challenging to think in terms of delayed branches. For the engineer using higher-level languages, compilers are quite effective at re-ordering code to employ delayed branches.

The SHARC also provides “traditional” (non-delayed) branches. These branches take three clock cycles since they reload the pipeline. Having a non-delayed branch instruction is a space-saving feature. When code cannot be re-ordered to use a delayed branch, the code must either use a delayed branch followed by two “no operation” instructions, or use a non-delayed branch. The non-delayed branch will take the same amount of time but use only a single word of storage space.

A *Harvard Architecture* allows a processor to fetch instructions from memory while concurrently reading or writing data to another memory location. This architecture uses two buses, a data memory bus and a program memory bus, and either two separate memories or a single dual-port memory (Figure 2). The compute stage of a processor, that may need to access data memory, need not compete with the execution unit, that must fetch the next instruction.

Many processors today employ an internal Harvard Architecture, meaning that the processor has two separate memories or caches integrated into the processor chip, but one single external bus. Processors using caches have a statistical performance benefit based upon cache hit rate. The SHARC is unique in that

it provides a Harvard Architecture by using a large internal memory capable of holding a large program and a large amount of data.

Engineers developing software for general purpose processors with an internal Harvard Architecture need not be aware of the architecture, other than to know that cache must be enabled for this to be of benefit. However, SHARC developers must be aware of the internal memory organization and the Harvard Architecture to take full advantage of this feature. A SHARC software developer determines where code and data reside at link time. The SHARC can access program and data memory concurrently if the linker places code in one block and data in the other block.

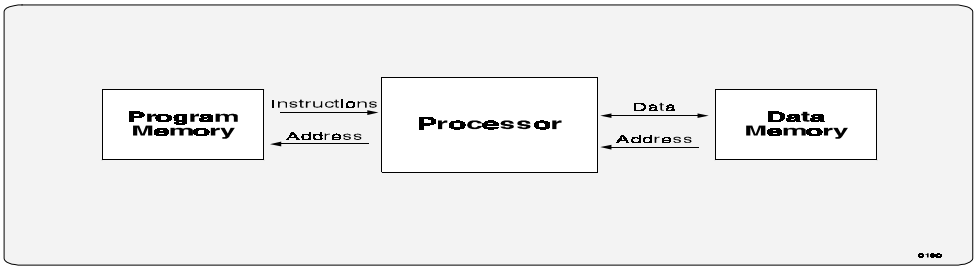

Figure 2. The use of separate busses for program memory and data within a Harvard Architecture allow the processor to fetch an instruction while reading or writing to data memory.

Cache. Caches allow a processor to maximize performance by reducing the probability that the processor will have to wait for slower external memory accesses. The SHARC incorporates a small program memory cache, but its behavior is more deterministic than the cache sported by other processors. Even though the SHARC uses a relatively low clock rate (40 MHz), its Harvard Architecture and large internal memory allow it to keep its execution unit busy without the aid of a cache. The SHARC uses its cache only when the processor executes an instruction that accesses data stored within program memory space. Such an access uses the program memory bus, competing with instruction fetch cycles. Instructions causing program memory access conflicts are cached. Subsequent fetch of these instructions comes from the cache. This allows the SHARC to fetch an instruction while accessing both program and data memory. Combining this with zero overhead looping allows the SHARC to move data between program and data memory at a steady-state rate of one word per clock cycle; a performance figure unattainable in many other architectures.

As is true with other processors, SHARC programmers need not know much about the cache, other than how to enable it, in order to benefit from it.

II. DSP Specific Performance Features.

A large number of DSP algorithms have repetitive computation on fixed length vectors. The inner product of two vectors can be used to determine energy, to compute filter results, compute a partial result for a matrix product, or determine correlation. An architecture that efficiently handles a small loops such as the inner product will provide superior performance for a wide variety of DSP algorithms.

Consider the following C code fragment that computes the inner product of two vectors, x and y, each having LENGTH elements. Depending upon the target machine, this code will compile into a loop having anywhere from two to eight instructions. The processor having the lowest product of instruction count and

instruction execution time will provide the best performance. As the vector length becomes large, the ratio of time spent in such loops to time spent outside the loops becomes even more significant.

```
result = 0;
for (n = 0; n < LENGTH; n++)
      result += x[n] * y[n];
```

The SHARC incorporates the following additional features aimed at optimizing such a loops. These features are DSP specific. That is, when applied to a non-DSP application, they might not provide a significant performance gain. These features are the following:

High-Speed Floating Point Capability. The SHARC supports both fixed and floating point computation. No additional clock cycles are required for floating point computation. Fixed and floating point SHARC instructions that provide an advantage in many DSP algorithms include MIN, MAX, and ABS functions, scaling, averaging, and logical functions. Functions that compute seeds for square root and divide operations greatly improve performance of these operations.

Extended Floating Point. The SHARC floating point registers include an additional 8 bits of mantissa data. This reduces noise introduced by rounding and quantization. Data are automatically truncated and zero padded when moved between 32-bit memory and internal registers. Internal memory may be configured to store 32-bit data words, or a smaller number of 40-bit data words.

This additional accuracy is not sufficient for numerical or scientific algorithms requiring double precision. Rather, it represents a compromise between processor complexity and capability. A 40 bit floating point number provides a 32-bit mantissa. 32 bits provides a signal to noise ratio of 192 dB, far exceeding the sampling accuracy of any physical device.

Large, dual-port, two-block internal memory. SHARC internal memory has three unusual attributes[3]:

> *Size.* The large size of internal memory allows many complex functions to be performed on-chip. At 4 million bits (21060), the SHARC can contain 128k 32-bit words. Many devices provide on-chip memory for high speed computation, but the memory size can be significantly smaller, and moving data into and out of the internal memory can offset the advantages of using it. However, the SHARC memory is sufficiently large to be used as primary memory for many applications; the need to move data between internal and external memory is eliminated.
>
> *Configurable.* Each of the two blocks of SHARC internal memory is independently configurable as 32 bit words, a smaller number of 48-bit words, or a combination of the two. Each block can contain instructions, data, or both.
>
> *Dual-Block, Dual-Port.* This organization allows the SHARC core processor to fetch instructions while performing data memory accesses to fully exploit the advantage of the Harvard Architecture. The two ports of each block also permit concurrent I/O at the maximum memory bandwidth without interfering with computation.

Multiply/Accumulate Instructions. Like most digital signal processors, the SHARC can compute a product and add the product to a running total in a single clock cycle. The accumulator registers can be floating point or extended (80-bit) fixed point. The SHARC includes instructions that multiply and accumulate while adding, subtracting, or averaging data in two other registers. These multifunction instructions give the SHARC its 120 megaflop rating at 40 MHz.

Zero overhead looping. The SHARC incorporates zero-overhead looping[4]. A single instruction outside the loop performs loop set-up by informing the SHARC that it will be executing a loop. The instruction

includes the iteration count and termination condition. This instruction causes the machine to keep the pipeline full during loop execution and test the termination condition in parallel with execution of loop contents. Thus, clock cycles that would normally be consumed by these loop management functions are eliminated.

Data Address Generators. The SHARC incorporates two Data Address Generators (DAGs). These are integer computation units dedicated to the management of index registers. Using the addressing modes provided by the DAGs, the SHARC can fetch a value, and update the index value using an arbitrary stride. If the updated index value exceeds a limit, the DAG adjusts the index so that it wraps. This occurs in the same clock cycle as the read or write operation.

These capabilities allow the DAGs to support the following operations[5]:

> *Circular Buffering.* Many DSP applications operate on vectors of samples. The next input vector is derived from the last input vector by shifting all elements of the last vector. The oldest entry in the vector is shifted out and a new entry is shifted in. Rather than actually move the data, circular buffers are employed. By updating an index modulo the buffer length, the oldest entry can be conveniently replaced by the newest entry (Figure 3).
>
> *Bit Reverse Addressing.* Performance of the Fast Fourier Transform (FFT) is often critical to DSP applications; special hardware features that enhance performance are justified. One requirement during FFT computation is bit-reversed addressing. In bit-reversed addressing, the bit pattern of a vector index value is reversed. The SHARC can automatically perform this bit reversal while accessing data.

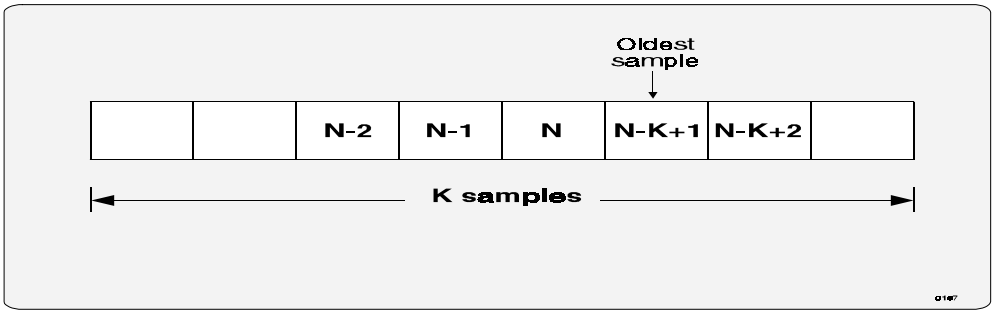

Figure 3. A circular buffer of length k. The next input sample, N+1, will overwrite the oldest sample, N-K+1, and the pointer to the oldest will advance to the right, wrapping around when the end of the buffer is reached.

Long Instruction Word. The SHARC uses a 48-bit instruction word. This wide instruction can incorporate fixed or floating point compute instructions, along with transfer of control instructions or data read/write instructions.

The following examples illustrate use of these features to maximize DSP performance.

Example 1. Inner Product of Two Vectors.

Consider the code fragment that computes the inner product of two vectors. An assembly language implementation of this loop for an arbitrary machine might look like this:

```
      Set result to 0
top:  load x[n]
      multiply by y[n]
      add to result
      increment n
      compare n against LENGTH
      if < jump to top
```

This code takes 6 clock cycles per loop iteration, plus additional cycles for pipeline reloading at the bottom of the loop. The above example assumes that both x and y can be indexed by n, and that the processor can multiply the contents of a register by the contents of a memory location. Using the same example and taking full advantage of the SHARC's looping and addressing features gives a single instruction loop. While the SHARC has a lower clock frequency than many general purpose processors, the one instruction loop required for this computation gives the SHARC a tremendous performance advantage:

```
/*
 *  Compute the inner product of two vectors, _x and _y.
 *  Vector length is the constant LENGTH.
 *  Result is placed in mrf.  The modify registers
 *  m6 and m14 are assumed to contain 1.
 */

#define LENGTH    128

      mrf = 0;                /* result register        */
      i4  = _x;               /* i4 indexes the x array */
      i12 = _y;               /* i5 indexes the y array */
      f0  = dm (i4, m6);      /* get first x value      */
      f1  = pm (i12, m14);    /* get first y value      */

      lcntr = LENGTH, do (pc, 1) until lce;
          mrf = mrf+f0*f1, f0 = dm(i4,m6), f1 = pm(i12,m14);
```

The construct `dm(i4,m6)` fetches data from the location contained in the i4 register, and then updates i4 by adding m6 to the register. The `lcntr =…` instruction sets up a loop that executes LENGTH times. The loop set-up instruction is executed once. The loop contents, one instruction in this example, are executed LENGTH times, keeping the pipeline full. When the loop ends, mrf will contain the result, and f0 and f1 will contain the values from memory that immediately follow the vectors.

Example 2. Data Movement.

The SHARC can read from program memory and write to data memory (or visa versa) in a single clock cycle. Together with zero overhead looping, we can write a program that moves data at a rate of one word per clock cycle. An interesting feature of the SHARC allows this program to execute at this rate when moving data between internal memory blocks 0 and 1. Whether either block actually contains instructions or not is immaterial; what makes this program fast is that it will use the program memory bus to read data, the data memory bus to write data, and the cache to fetch instructions while the program memory bus is busy:

```
/*
 *  Move memory from where i12 points to
 *  where i4 points.  Move r1 number of words.
 *  Assumes m6 and m14 contain a '1'.
 */

r0 = pm (i12, m14);     /* fetch first word into r0 */
lcntr = r1, do (pc, 1) until lce;
      dm (i4, m6) = r0, r0 = pm (i12,m14);
```

The single instruction within the loop writes the "old" value of register r0 to memory while reading a new value into r0. Because of this, the program must first "prime the pump" by loading the first value into r0.

Example 3. IXLibs-21k.

IXLibs-21k is a library of mathematical functions for the 21000 family of floating point DSPs. The library includes signal processing functions, math functions, vector functions, statistical functions, and some miscellaneous functions. These functions have been optimized to take advantage of all SHARC features. As a result, functions like vector add, dot product, vector absolute value, etc., require one or two clock cycles per vector element[6]. All IXLibs-21k functions are C callable, and run on the SHARC.

III System Integration: Input/Output Features of the SHARC

A powerful DSP engine is of little benefit if spends much of its time waiting for input and output operations. Realizing this, the designers of the SHARC included a powerful, autonomous I/O capability into the SHARC. This capability is provided by the SHARCs Input/Output Processor (IOP). The IOP is an integral part of the SHARC chip, and operates independently of the core processor (the core being the traditional CPU within the SHARC). The IOP incorporates the following interfaces:

An *external port*, that can access memory or devices external to the SHARC

Six *link ports*, that provide high-speed (40 megabytes per second) communication with other SHARCs.

Two high-speed *serial ports*, that can perform word packing and unpacking, companding, and TDM demultiplexing.

Ten *Direct Memory Access (DMA) channels*, that move data between any of the above ports and internal memory. The DMA channels support chaining, that allows a channel to reload its control registers from memory and start a new transfer at the conclusion of a transfer.

Because the IOP has its own port into each block of the internal dual port memory, the IOP may access internal memory at full bandwidth (40 megawords per second) without impeding core processor access.

External ports are managed by either the core processor or a DMA channel. Ixthos produces SHARC-based VME cards that have sites for high speed I/O mezzanines. Driver software provided by Ixthos uses the DMA channels to move data between the SHARC internal memory and the mezzanines via the external ports. Doing this allows the core processor to operate on the last input buffer while the DMA fills the next input buffer. The independent operation of the DMA channel and dual porting of internal memory blocks allow I/O to take place at up to 40 million words per second without impeding processor performance.

Link ports may be accessed by either DMA channels or the core processor. The greatest challenge to using link ports is determining that side is allowed to transmit. Ixthos has developed a low-latency, bidirectional link port protocol that allows processors to send variable length messages. The software uses a proprietary low-latency software token passing scheme to manage the direction of link port traffic. The software also provides message queuing on both the send and receive sides. The software automatically queues messages if the link port is busy, and works off the queue as each message transfer completes.

For applications that do not require bidirectional link port transfers, the software also supports a faster, single-direction, transfer mode. This mode also provides message queuing, but the absence of token management provides even greater throughput. Link utilization in excess of 98% is achieved using the 2X clock rate mode (40 megabytes per second) and message lengths of 512 words.

Difficulties arose in the development of link port software for the following reasons:

1. Early revisions of the SHARC (pre rev 2.0) exhibited problems whenever the core wrote to the link control register. Writing to this register is necessary to start or complete link port data transfers.

2. When using the link ports with DMA transfers, it is important to remember that the DMA will generate an interrupt when the last word has been transferred into the link port buffer. Because this buffer is a FIFO holding two words and because the receiver can throttle the

sender using flow control signals provided in the link port, completion of the DMA transfer can precede completion of the link port transfer by several clock cycles. Therefore it is not appropriate to reprogram the link port from the DMA interrupt service routine until the interrupt service routine determines that the link port buffer is empty.

3. The SHARC latches interrupts internally. While servicing a specific interrupt, the SHARC will not latch that interrupt until the interrupt service routine exits. This creates a problem when a DMA interrupt service routine starts another DMA transfer that completes before the service routine exits. Fortunately, the SHARC instruction set provides a way around this.

The serial port capabilities are equally powerful. A serial port can bring in a stream of data, maintain frame synchronization, and extract specific samples from a frame. The serial port can then expand these samples using mu-law or A-law companding. Serial port output operations can compress data using mu-law or A-law companding. These capabilities are of little value for most embedded applications. However, they allow management of time-division multiplexed (TDM) data streams commonly used in telephony applications without consuming core processor cycles.

IV. Clusters.

Up to six SHARCs may be tied together to form a cluster. Any SHARC can access the internal memory of any other SHARC in a cluster. A broadcast feature allows a SHARC to write to the same location of all SHARCs in the cluster using a single write cycle. Problems that can be partitioned and allocated to multiple processors can take advantage of clustering by sharing status and data.

Processors may use round-robin or priority bus arbitration to access the bus. An internal, programmable bus time out counter prevents a processor from dominating the cluster bus. A bus lock capability allows a processor to intentionally dominate the cluster bus. This facilitates atomic operations within shared memory. Ixthos has employed these features in the shared memory malloc functions. These functions allow all processors on a board to allocate areas of memory of arbitrary size from a single shared memory segment.

Ixthos has organized its VME boards as two clusters of two SHARCs, two clusters of four SHARCs, or four clusters of four SHARCS.

V. Other Performance Considerations.

High-Level Language Support.

As with any processor, the objective is to minimize the storage requirements and maximize performance. Understanding the costs and adjusting code to optimize the costs requires knowing the machine well. It is relatively easy to get any application running in C. Optimizing performance may require implementing some functions in assembly language.

The SHARC C compiler is efficient. It performs reasonable optimization and register allocation. However, compilers cannot analyze dynamic execution requirements of code. Therefore they can optimize in such a way that the path most taken is not optimal. Also, maintaining the C program stack and stack frame require several instructions of prologue and epilogue code for each function, usually at least three instructions to call a function, and three more to return.[7] Expect at least 6 clock cycles to call a simple function.

The SHARC C function libraries support interrupts. The SHARC interrupt latency is very low, typically 3 clock cycles. An interrupt service routine must save and restore the context of the machine. The C

compiler and the interrupt service shells expect the C runtime environment to be maintained. Applications using both C and assembly language must be aware of this. Saving and restoring the machine context adds in excess of 100 clock cycles to the interrupt service time.

The SHARC has a large set of registers. Most registers have “alternates” that are selected by changing bits in the MODE 1 register. Changing these bits provides a quick way to provide an alternate context. This approach does not lend itself to interrupt nesting. Ixthos has developed C compatible code that allows an application to switch to the alternate register set when servicing an interrupt, while maintaining the C runtime environment. This is targeted for those applications implemented in C that require low interrupt latency but do not require nesting of prioritized interrupts.

VMEbus Access.

Ixthos VME boards do not tie the SHARC processors directly to the VMEbus. A single VME access, including bus arbitration, can require close to a microsecond depending upon the performance of the slot 1 arbiter card. If other cards are contending for the bus, times can greatly exceed one microsecond. In one microsecond, the SHARC can perform 120 floating point computations.

Instead, Ixthos isolates the SHARCs from the VMEbus using a board controller that manages VME transfers on behalf of the SHARCs. Applications use straightforward VME function calls to task the board controller to perform VME transfers, including block transfers and read/modify/write cycles.

Ixthos boards have multiple global memory mezzanine sites. Memory mezzanine options span the range from 128k words of static RAM to 16 megawords (64 megabytes) of dynamic RAM. These mezzanines are directly accessible from VMEbus, and are connected to the clusters through a crossbar switch (figure 4). As one cluster accesses one memory mezzanine, another cluster or the VMEbus can access another mezzanine.

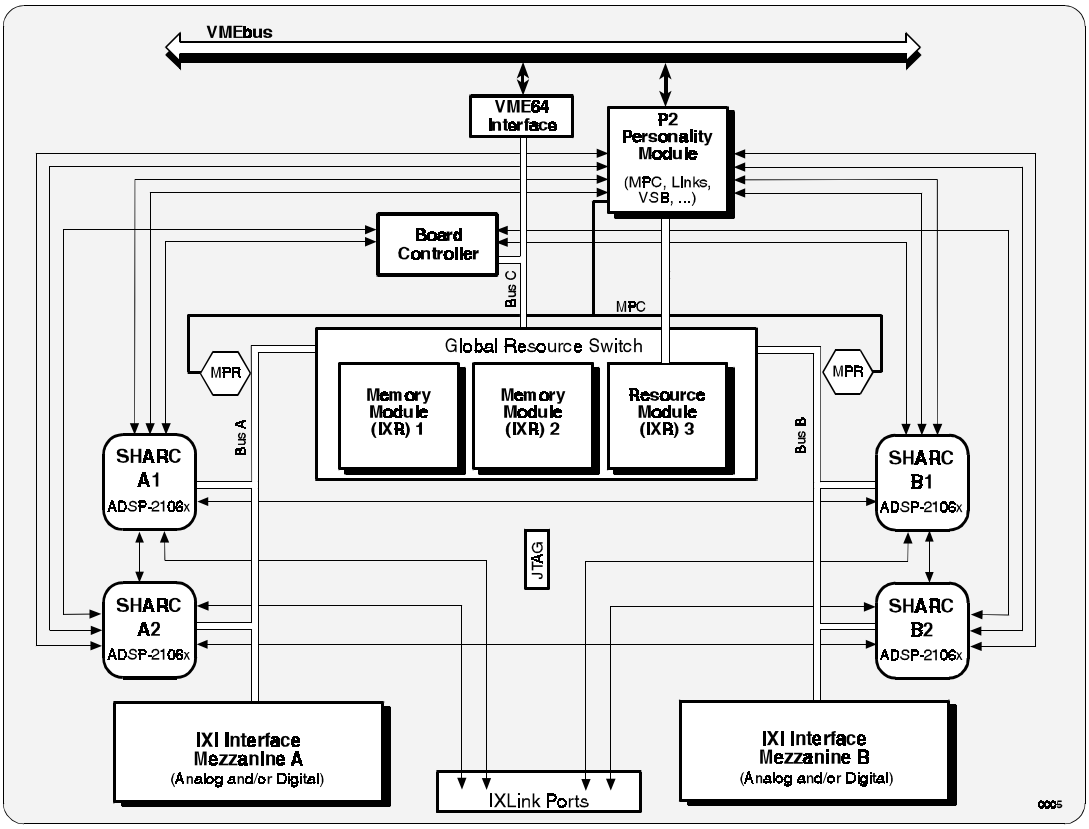

Figure 4. The Global Resource Switch is a crossbar switch allowing any cluster or VME bus to access any of the three IXR memory mezzanines.

VI. Summary

When optimizing SHARC software performance, take the following steps:

1) Examine memory partitioning carefully. Take advantage of the dual-port, two-block internal memory. Failure to do so will result in a program that generates correct results but has half the performance.

2) Optimize the innermost loop. Identify the 10% of the code that executes 90% of the time. If possible, implement primitive functions in assembly, taking advantage of looping, DAG capabilities, and other special instructions.

3) Consider using DMAs and chaining, when performing link port or serial I/O. This will free up the core processor. Ixthos provides library functions that perform bidirectional data transfer over link ports using the DMAs. These libraries also support message queuing. This combination of features allows a single SHARC processor to concurrently input data, operate on previously received data, and transmit results computed on another message. The full overlap of input, output, and processing is accomplished using bi-directional link port drivers and dynamic message queuing with minimal overhead (< 500 clock cycles fixed core overhead). Unidirectional drivers provide fixed length message passing using < 200 clock cycles core overhead. Further refinements in library functions will continue to reduce these numbers.

4) Examine interrupt service latency requirements carefully. Is nesting required? Can alternate registers be used?

[1] ADSP-2106x SHARC User's Manual, Analog Devices Inc., First Edition, Copyright 1995. p. 1-3. Diagram and all substantial architectural details regarding SHARC internal architecture are attributable to this document.

[2] ibid., pp. 1-1 to 1-18, 3-2 to 3-4, and 3-10 to 3-12.

[3] ibid., pp. 5-1 to 5-5.

[4] ibid., pp. 3-13 to 3-20.

[5] ibid., pp. 4-1 to 4-10.

[6] IXLibs-21k User's Manual, Ixthos Inc. and LightSpray, Inc., Copyright 1995. pp. 238-257.

[7] ADSP-21000 Family C Tools Manual, Analog Devices Inc., Third Edition, Copyright 1995. pp. 4-9 to 4-10.